Java Lambda Expressions

Jakob Jenkov
Last update: 2015-03-10

Java lambda expressions are new in Java 8. Java lambda expressions are Java's first step into functi programming. A Java lambda expression is thus a function which can be created without belonging t any class. A lambda expression can be passed around as if it was an object and executed on deman

Java Lambdas and the Single Method Interface

Functional programming is very often used to implement event listeners. Event listeners in Java are o defined as Java interfaces with a single method. Here is a fictive single method interface example:

```
public interface StateChangeListener {

    public void onStateChange(State oldState, State newState);

}
```

This Java interface defines a single method which is called whenever the state changes (in whatever i being observed).

In Java 7 you would have to implement this interface in order to listen for state changes. Imagine you have a class called `StateOwner` which can register state event listeners. Here is an example:

```
public class StateOwner {

    public void addStateListener(StateChangeListener listener) { ... }

}
```

In Java 7 you could add an event listener using an anonymous interface implementation, like this:

```
StateOwner stateOwner = new StateOwner();

stateOwner.addStateListener(new StateChangeListener() {

    public void onStateChange(State oldState, State newState) {
        // do something with the old and new state.
    }
});
```

In Java 8 you can add an event listener using a Java lambda expression, like this:

```
StateOwner stateOwner = new StateOwner();

stateOwner.addStateListener(
    (oldState, newState) -> System.out.println("State changed")
);
```

The lambda expressions is this part:

```
(oldState, newState) -> System.out.println("State changed")
```

The lambda expression is matched against the parameter type of the `addStateListener()` method's parameter. If the lambda expression matches the parameter type (in this case the `StateChangeListe` interface) , then the lambda expression is turned into a function that implements the same interface a that parameter.

Java lambda expressions can only be used where the type they are matched against is a single meth interface. In the example above, a lambda expression is used as parameter where the parameter type was the `StateChangeListener` interface. This interface only has a single method. Thus, the lambda expression is matched successfully against that interface.

Matching Lambdas to Interfaces

A single method interface is also sometimes referred to as a *functional interface*. Matching a Java lam expression against a functional interface is divided into these steps:

- Does the interface have only one method?
- Does the parameters of the lambda expression match the parameters of the single method?
- Does the return type of the lambda expression match the return type of the single method?

If the answer is yes to these three questions, then the given lambda expression is matched successfu against the interface.

Lambda Type Inference

Before Java 8 you would have to specify what interface to implement, when making anonymous inter implementations. Here is the anonymous interface implementation example from the beginning of this text:

```
stateOwner.addStateListener(new StateChangeListener() {

    public void onStateChange(State oldState, State newState) {
        // do something with the old and new state.
    }
});
```

With lambda expressions the type can often be *inferred* from the surrounding code. For instance, the interface type of the parameter can be inferred from the method declaration of the `addStateListene` method (the single method on the `StateChangeListener` interface). This is called *type inference*. The compiler infers the type of a parameter by looking elsewhere for the type - in this case the method definition. Here is the example from the beginning of this text, showing that the `StateChangeListene` interface is not mentioned in the lambda expression:

```
stateOwner.addStateListener(
    (oldState, newState) -> System.out.println("State changed")
);
```

In the lambda expression the parameter types can often be inferred too. In the example above, the compiler can infer their type from the `onStateChange()` method declaration. Thus, the type of the parameters `oldState` and `newState` are inferred from the method declaration of the `onStateChange(` method.

Lambda Parameters

Since Java lambda expressions are effectively just methods, lambda expressions can take parameter just like methods. The `(oldState, newState)` part of the lambda expression shown earlier specifies parameters the lambda expression takes. These parameters have to match the parameters of the method on the single method interface. In this case, these parameters have to match the parameters the `onStateChange()` method of the `StateChangeListener` interface:

```
public void onStateChange(State oldState, State newState);
```

As a minimum the number of parameters in the lambda expression and the method must match.

text), but in many cases you don't need them.

Zero Parameters

If the method you are matching your lambda expression against takes no parameters, then you can w
your lambda expression like this:

```
() -> System.out.println("Zero parameter lambda");
```

Notice how the parentheses have no content in between. That is to signal that the lambda takes no parameters.

One Parameter

If the method you are matching your Java lambda expression against takes one parameter, you can w
the lambda expression like this:

```
(param) -> System.out.println("One parameter: " + param);
```

Notice the parameter is listed inside the parentheses.

When a lambda expression takes a single parameter, you can also omit the parentheses, like this:

```
 param -> System.out.println("One parameter: " + param);
```

Multiple Parameters

If the method you match your Java lambda expression against takes multiple parameters, the parame
need to be listed inside parentheses. Here is how that looks in Java code:

```
(p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);
```

Only when the method takes a single parameter can the parentheses be omitted.

Parameter Types

Specifying parameter types for a lambda expression may sometimes be necessary if the compiler ca
infer the parameter types from the functional interface method the lambda is matching. Don't worry, t
compiler will tell you when that is the case. Here is a Java lambda parameter type example:

```
(Car car) -> System.out.println("The car is: " + car.getName());
```

As you can see, the type (`Car`) of the `car` parameter is written in front of the parameter name itself, jus
like you would when declaring a parameter in a method elsewhere, or when making an anonymous implementation of an interface.

Lambda Function Body

The body of a lambda expression, and thus the body of the function / method it represents, is specifi
to the right of the `->` in the lambda declaration: Here is an example:

```
 (oldState, newState) -> System.out.println("State changed")
```

If your lambda expression needs to consist of multiple lines, you can enclose the lambda function bo
inside the `{ }` bracket which Java also requires when declaring methods elsewhere. Here is an examp

```
 (oldState, newState) -> {
    System.out.println("Old state: " + oldState);
    System.out.println("New state: " + newState);
  }
```

Returning a Value From a Lambda Expression

You can return values from Java lambda expressions, just like you can from a method. You just add a
return statement to the lambda function body, like this:

```
 (param) -> {
    System.out.println("param: " + param);
    return "return value";
  }
```

```
(a1, a2) -> { return a1 > a2; }
```

You can write:

```
(a1, a2) -> a1 > a2;
```

The compiler then figures out that the expression `a1 > a2` is the return value of the lambda expressio
(hence the name lambda *expressions* - as expressions return a value of some kind).

Lambdas as Objects

A Java lambda expression is essentially an object. You can assign a lambda expression to a variable
pass it around, like you do with any other object. Here is an example:

```
public interface MyComparator {

    public boolean compare(int a1, int a2);

}
```

```
MyComparator myComparator = (a1, a2) -> return a1 > a2;

boolean result = myComparator.compare(2, 5);
```

The first code block shows the interface which the lambda expression implements. The second code
block shows the definition of the lambda expression, how the lambda expression is assigned to varia
and finally how the lambda expression is invoked by invoking the interface method it implements.

Next: **Java Exercises**

G+ Share Tweet

Jakob Jenkov